# 18

# Blockchain Privacy

In this chapter, we'll cover privacy, which is an important topic. Data privacy or information privacy is not a new concept. Data privacy refers to the protection of personal and sensitive information and the rights of individuals to control how their personal information is collected, stored, used, and shared. While privacy has always been important, with the advent of the internet and online services, protecting personal information has become even more crucial than ever. In the established digital world, like the internet and other online services, there are many laws and regulations, along with strict information security policies and technical controls protecting users.

However, in the blockchain world, there is a sheer lack of such rigor so far. Achieving the same level of security that the typical digital world enjoys is still an issue that needs to be addressed. In this chapter, we'll see what privacy in the context of blockchain is and how we can introduce it using different protocols. Along the way, we'll look at the following topics:

- Privacy and its types
- Layer 0, Layer 1, and Layer 2 protocols for privacy on blockchain
- Zero-knowledge proofs, their various types, polynomial commitment schemes, and relevant protocols
- A practical example

So, let's begin; first we'll discuss privacy, why it's important, and its various types.

## Privacy

Privacy in blockchain can be divided into two main categories based on the type of service required. These categories are *the anonymity of the users* and *the confidentiality of the transactions*. Anonymity is concerned with hiding the sender's or receiver's identity, whereas confidentiality addresses the requirements of hiding transaction values.

The fundamental reason why blockchains are not privacy-preserving is that every transaction in a blockchain needs to be verified and executed by every participant on the network. While this property of executing all transactions by all nodes gives blockchains their powerful integrity property and execution guarantee, this is also a weakness.

This weakness is the result of inherent transparency in blockchains. As all transaction data, including account details, inputs, outputs, and states are visible to anyone on the blockchain, privacy cannot be preserved.

One solution that comes to mind is that we could somehow encrypt the data, but if the values are hidden, then the transactions cannot be verified. The key requirement is to somehow combine public verifiability and confidentiality so that even if the data is encrypted in some form, it still can be publicly verified.

Let's now explain anonymity and confidentiality, before we delve deeper into what can we do to solve this problem.

## Anonymity

Anonymity is desirable in situations where the identity of users is required to be hidden from other participants on a network. This can be due to regulatory requirements, enterprise requirements, or just due to the sensitive nature of the transactions. For example, in a business network, it might be desirable to hide the dealings between different participants on the network from other competitors to avoid friction or unjustified business competitive advantage.

The properties of *unlinkability* and *being untraceable* are used to achieve anonymity. The property of untraceability allows us to hide the trace of a transaction from one party to another in a network. On the other hand, unlinkability means that an observer is unable to deduce the link between transactions and their participants, or the relationships between transaction participants (senders and receivers). This means that if a third party looks at the transaction flow, they cannot see which entities are engaged in transactions with which other entities.

There are several blockchains that support privacy inherently and are built into the design of a system. The most popular is Zcash, which uses **zero-knowledge proofs** (**ZKPs**) to achieve anonymity. Other examples include Monero (`https://web.getmonero.org`), which makes use of ring signatures to provide anonymity services.

## Confidentiality

Confidentiality is an absolute requirement in many industries such as finance, law, and health. Similarly, the privacy of transactions is a much-desired property of blockchains. However, due to its very nature, especially in public blockchains, everything is transparent, thus inhibiting its usage in various industries where privacy is of paramount importance, such as finance, health, and many others.

When we think about confidentiality we can divide it into two further optional requirements, **conditional privacy** and **selective disclosure.** Conditional privacy means that a system should have the ability to conditionally make data visible to a third party such as auditors, but keep the data hidden from all other parties except those who are privy to the transactions. Note that **unconditional privacy** is also not good in practice, because if everything is hidden from everyone it can permit criminal activities such as money laundering and terrorist financing, without anyone knowing about it. So, there is a need, especially in financial applications of blockchain, to ensure that in some cases a third party such as a regulator or an auditor is able to access and view data to ensure compliance with regulatory requirements.

The other confidentiality type is **selective disclosure**, where a system should have the ability to selectivity share some part of the data, such as age only, from a larger dataset containing personal information. Another variation of this is called **range proof**, which is the ability to prove that some data is in a range, for example, my ability to prove that I am older than 18 years of age or my salary range is between $10,000 and $20,000.

We will now look at some techniques that we can use to achieve privacy.

## Techniques to achieve privacy

As the blockchain is a public ledger of all transactions and is openly available, it becomes easy to analyze it. When combined with traffic analyses, transactions can be linked back to their source IP addresses, thus possibly revealing a transaction's originator. This is a big concern from a privacy point of view.

In the Bitcoin domain it is a recommended and common practice to generate a new address for every transaction, which allows some level of unlinkability. However, this is not enough, and various techniques were developed and successfully used to trace the flow of transactions throughout the network and link them back to their originator. These techniques analyze blockchains by using transaction graphs, address graphs, and entity graphs, which facilitate linking users back to the transactions, thus raising privacy concerns.

These techniques are based on the idea that every transaction on the Bitcoin network (or any public blockchain) is publicly recorded and can be linked to other transactions, addresses, and entities. For example, transaction graph analysis involves creating a graph of all transactions and identifying patterns of transactions between addresses. Address graph analysis involves creating a graph of all addresses and their associated transactions. This type of analysis can help identify the relationships between different addresses and the flow of funds between them.

The techniques mentioned earlier can be further enriched by using publicly available information (for example, public internet forum users' Bitcoin addresses) about transactions and linking them to the actual users. There are open source block parsers available that can be used to extract transaction information, balances, and scripts from the blockchain database.

In this section, we'll describe different techniques to achieve privacy, including anonymity and confidentiality.

Similar to scalability solutions, we can also divide privacy solutions into three categories, based on the layer within the blockchain architecture stack in which they operate:

- **Layer 0** methods, or network layer methods, where the mechanism to achieve privacy operates at a network level.
- **Layer 1** methods, also called on-chain methods, where the blockchain protocol itself is enhanced to achieve privacy.
- **Layer 2** methods, or off-chain methods, where mechanisms that exist outside of the main blockchain are used to achieve privacy on the blockchain.

## Layer 0

These solutions are network layer-level methods that provide anonymity by using TOR and I2P. These methods allow us to hide the identities of the parties involved.

### Tor

**Tor**, **The Onion Router**, is a software that enables anonymous communications. More information on Tor is available here: `https://www.torproject.org`. We can use Tor to enable anonymous communication in cryptocurrency blockchain networks. Tor is a common choice to enable anonymous communication and has been used in Monero, Verge, and in **Bitcoin** to provide anonymity.

For Bitcoin, see `https://en.bitcoin.it/wiki/Tor`. Monero can also be run with Tor (`https://web.getmonero.org/`). Verge is another example of cryptocurrency making use of Tor for IP obfuscation.

### I2P

**I2P**, the **Invisible Internet Project**, is an anonymous network built on the internet. It enables censorship-resistant peer-to-peer communication. It is used in Monero to provide anonymity services. More information on I2P is available here: `https://geti2p.net/en/`. Monero and Zcash support anonymous transactions. Monero makes use of ring signatures, stealth addresses, and confidential transactions, whereas Zcash uses zk-SNARKs.

As many of the techniques described in the following sections can be used at Layers 1 and 2—for example, ZKPs—we are not distinguishing between these layers. Instead, we'll only list these solutions and discuss what they can be used for. Furthermore, solutions for anonymity and confidentiality also overlap and techniques used to achieve confidentiality are also more or less applicable to achieving anonymity. We do, however, segregate Layer 0, the *network layer*, and we will describe some of the network-level approaches to achieve anonymity, before moving on to introduce a range of privacy mechanisms that are applicable to Layers 1 and 2.

## Indistinguishability obfuscation

This cryptographic technique may serve as a silver bullet to all privacy and confidentiality issues in blockchains, but the technology is not yet ready for production deployments. **Indistinguishability obfuscation** (**IO**) allows for code obfuscation, which is a very ripe research topic in cryptography. If applied to blockchains, IO can serve as an unbreakable obfuscation mechanism that will turn smart contracts into a black box where the behavior of the obfuscated code is indistinguishable. In simpler words, the inner functionality of the smart contract is totally hidden.

The key idea behind IO is what's called by researchers a *multilinear jigsaw puzzle*, which basically obfuscates program code by mixing it with random elements, and if the program is run as intended, it will produce the expected output. However, any other way of executing it would make the program output random garbage data.

In other words, IO makes it computationally infeasible for an attacker to distinguish between two different program executions, even if the attacker has complete access to the program's code and input/output behavior.

In other words, IO provides a way to protect the privacy of a program's implementation, making it impossible for an attacker to determine the exact details of how the program executes or what inputs it was given.

This research paper is available at `https://eprint.iacr.org/2013/451.pdf`.

## Homomorphic encryption

This type of encryption allows operations to be performed on encrypted data. Imagine a scenario where the data is sent to a cloud server for processing. The server processes it and returns the output without knowing anything about the data that it has processed. This is also an area ripe for research and fully homomorphic encryption, which allows all operations on encrypted data, is still not fully deployable in production; however, major progress in this field has already been made. Once implemented on blockchains, it can allow processing encrypted transactions without requiring to decrypt them, which will inherently allow the privacy and confidentiality of transactions.

For example, the data stored on the blockchain can be encrypted using homomorphic encryption, and computations can be performed on that data without the need for decryption, thus providing privacy service on blockchains. This concept has also been implemented in a project named *Enigma*, which is available online at `https://www.media.mit.edu/projects/enigma/overview/`, by MIT's Media Lab. Enigma is a peer-to-peer network that allows multiple parties to perform computations on encrypted data without revealing anything about the data. The research is available at `https://crypto.stanford.edu/craig/`.

## Secure multiparty computation

The concept of secure multiparty computation is not new and is based on the notion that data is split into multiple partitions between participating parties, under a secret sharing mechanism, which then does the actual processing on the data without the need to reconstruct data on a single machine.

The output produced after processing is also shared between the parties. Multiple parties carry out the computation mutually without revealing their inputs. Only the output of the computation is revealed.

A secure multiparty computation platform called Enigma was proposed in 2015. The paper is available here: `https://web.media.mit.edu/~guyzys/data/enigma_full.pdf`.

## Trusted hardware-assisted confidentiality

Trusted computing platforms can be used to provide a mechanism by which confidentiality of a transaction can be achieved on a blockchain, for example, by using Intel **Software Guard Extensions** (**SGX**), which allows code to be run in a hardware-protected environment called an **enclave**. Once the code runs successfully in the isolated enclave, it can produce a proof, called a **quote**, which is attestable by Intel's cloud servers.

It is a concern that trusting Intel will result in some level of centralization and is not in line with the true spirit of blockchain technology. Nevertheless, this solution has its merits and, in reality, many platforms already use Intel chips anyway, so trusting Intel may be acceptable by some in some cases.

If this technology is ap1plied to smart contracts, then once a node has executed the smart contract, it can produce a proof of correctness, thus proving successful execution, and other nodes will only have to verify it. This idea can be further extended by using any **Trusted Execution Environment** (**TEE**), which can provide the same functionality as an enclave and is available even on mobile devices with **Near Field Communication** (**NFC**) and a secure element. For example, Ekiden is a platform that makes use of Intel SGX's TEE to run smart contracts while preserving confidentiality. More information on Ekiden is available here: `https://arxiv.org/pdf/1804.05141`.

## Mixing protocols

A **mixing protocol**, or mixer, is a service that allows users to preserve their privacy by mixing their coins with other users.

These schemes are used to provide anonymity to cryptocurrency transactions. In this model, a mixing service provider (an intermediary or a shared wallet) is used. Users send coins to this shared wallet as a deposit, and then the shared wallet can send some other coins (of the same value deposited by some other users) to the destination. Users can also receive coins that were sent by others via this intermediary. This way, the link between outputs and inputs is no longer there, and transaction graph analysis will not be able to reveal the actual relationship between senders and receivers:

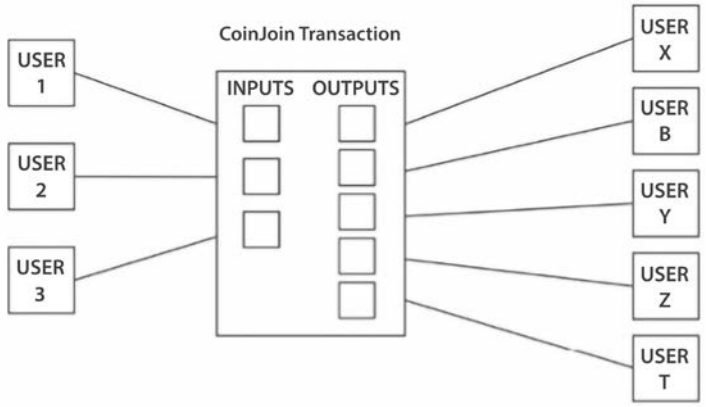


*Figure 18.1: Three users joining their transaction into a single larger Coinjoin transaction*

Coinjoin is one example of mixing protocols, where two transactions are joined together to form a single transaction while keeping the inputs and outputs unchanged. The core idea behind Coinjoin is to build a shared transaction that is signed by all participants. This technique improves privacy for all participants involved in the transactions.

**Coinjoin** is a technique that is used to anonymize **Bitcoin** transactions by mixing them interactively. The idea is based on forming a single transaction from multiple entities without causing any change in inputs and outputs. It removes the direct link between senders and receivers, which means that a single address can no longer be associated with transactions, which could lead to the identification of users.

Coinjoin needs to cooperate between multiple parties that are willing to create a single transaction by mixing payments. Therefore, it should be noted that, if any single participant in the Coinjoin scheme does not keep up with the commitment made to cooperate in creating a single transaction, by not signing the transactions as required, then it can result in a **DoS** (**Denial of Service**) attack.

In this protocol, there is no need for a single trusted third party. This concept is different from mixing a service, which acts as a trusted third party or intermediary between the Bitcoin users and allows the shuffling of transactions. This shuffling of transactions results in the prevention of tracing and linking payments to a particular user.

Various third-party mixing services are also available, but if a service is centralized, then it poses the threat of tracing the mapping between the senders and receivers. This is because the mixing service knows about all inputs and outputs. In addition to this, fully centralized miners pose the risk of the administrators of the service stealing the coins.

Various services, with varying degrees of complexity, such as CoinShuffle, Coinmux, and Darksend in Dashcoin, are available, which are based on the idea of Coinjoin (mixing) transactions. CoinShuffle is a decentralized alternative to traditional mixing services as it does not require a trusted third party.

Coinjoin-based schemes, however, have some weaknesses, most prominently the possibility of launching a **DoS** attack by users who committed to signing the transactions initially but now are not providing their signature, thus delaying or stopping joint transactions altogether.

## CoinSwap

CoinSwap is a privacy mechanism that is based on the idea of **atomic swaps**. Atomic swaps allow two parties to exchange coins without requiring a trusted third party. CoinSwap can also be used for cross-chain swaps. CoinSwap works by utilizing a third party in the transaction flow and also requires private communication channels between all parties. This way, the addresses of the sender and the receiver cannot be linked. This third party receives funds from the sender and sends them to the receiver.

The technique here is that the third party pays funds to the receiver by using a totally different source of funds, thus disconnecting the link between the sender and the receiver. This disconnection between the sender and the receiver results in providing an unlinkable transaction and thus privacy. The sender uses multi-signature transactions (usually 2 of 2—sender and third party) to allow transactions to be spent, while the receiver also requires multi-signature transactions (usually 2 of 2—receiver and third party) for transactions to be spent. In other words, the sender and the third party will sign the transaction output to send the Bitcoin to the third party, while the receiver and the third party will sign the transaction output to send the Bitcoin to the receiver.

CoinSwap uses hash-locked transactions where a pre-image of the hash is required to unlock the transaction. Using hash locked transactions prevents the third party from stealing the Bitcoin. More information on CoinSwap can be found at the following link, where it was originally proposed by Gregory Maxwell: `https://bitcointalk.org/index.php?topic=321228`.

## TumbleBit

The TumbleBit protocol was introduced in 2016. TumbleBit is fully compatible with the Bitcoin protocol. It is an anonymous, fast, and off-chain payments (unlinkability) protocol that allows parties to transfer funds via an untrusted third party or intermediary called a **tumbler**. In this protocol, even the tumbler is unable to deanonymize the payers and payees involved in a payment. It involves using two fair exchange protocols that prevent any malicious activity, such as cheating participants or the tumbler. TumbleBit relies on a protocol called **RSA puzzle solver**, which allows a payer to make payments to the tumbler. Unless tumbler solves this RSA puzzle, it cannot claim any Bitcoin paid by the payer. Another fair exchange protocol, called the **puzzle-promise** protocol, is used between the tumbler and the payee to claim the payment.

TumbleBit consists of three phases, as listed here:

1. Escrow phase, where all payment channels are set up.
2. Payments phase, where payers transfer funds.
3. Cash-out phase, where payers and payees close the payment channels.

More information on TumbleBit can be found here: `https://eprint.iacr.org/2016/575.pdf`. A proof of concept implementation of TumbleBit is available here: `https://github.com/BUSEC/TumbleBit`.

A recent innovation is the introduction of TumbleBit++, which is an improved mixing protocol based on TumbleBit. It provides anonymity and confidentiality of transaction amounts by combining confidential transactions and a centralized untrusted anonymous payment hub. More information regarding TumbleBit++ is available here: `https://link.springer.com/chapter/10.1007/978-3-030-31919-9_21`.

We will now discuss **Dandelion**, which provides inherent anonymity and is implemented by redesigning the network layer of the Bitcoin protocol.

## Dandelion

In addition to the approaches mentioned previously, a recent proposal called **Dandelion** has also been made. Dandelion (the improved version is called Dandelion++) is a proposal that aims to make transactions on a Bitcoin network untraceable. This protocol will allow anonymous transactions to occur on the Bitcoin network as opposed to pseudonymous transactions, where an adversary, by using network analysis methods, can trace the transaction back to its source node. Consequently, the adversary can then discover the original IP address of the transaction sender.

Deanonymization of Bitcoin users is a known problem, and a number of research papers are available on this topic. For example, the paper *Deanonymization of clients in Bitcoin P2P network* is available at `https://arxiv.org/pdf/1405.7418.pdf`.

We can say that Dandelion is a mechanism to provide inherent anonymity for Bitcoin transactions because, once implemented, the P2P layer of the network Bitcoin protocol is modified in such a way that tracing transactions back to their source node and IP would become extremely difficult.

A Bitcoin improvement proposal is available at `https://github.com/bitcoin/bips/blob/master/bip-0156.mediawiki`. The original Dandelion proposal paper is available at `https://arxiv.org/pdf/1701.04439.pdf`. The Dandelion++ research paper is available at `https://arxiv.org/pdf/1805.11060.pdf`.

In the Bitcoin network, the **epidemic flooding** mechanism (Gossip protocol) is used for transaction propagation. On top of this mechanism, there is a somewhat effective method called **diffusion**, which is used to provide some level of anonymity. In this protocol, each node introduces independent and exponential delays for spreading transactions to its neighbors. This scheme results in reducing the symmetry of the epidemic protocol, which makes network analysis difficult and unreliable. However, this scheme is predictable and hence does not provide sufficient anonymity guarantees.

Dandelion proposes a new but backward-compatible routing mechanism. In this protocol:

- First, a privacy graph (anonymity set) is constructed. This phase is called the **private graph construction** phase. It is a sub-graph of the existing Bitcoin P2P network, and each node selects a subset of its outbound peers in this phase. This is where the **random line of nodes** is selected.
- The messages (transactions) are then routed through this privacy graph during the **Stem** phase. This is where the message is propagated on a random line of nodes.
- Finally, there is the **Fluff** phase, where the messages are routed (broadcast) to the entire network by diffusion.

In summary, the dandelion protocol is composed of an **anonymity** phase and a **spreading** phase. In the anonymity phase, this protocol spreads a message over a random line for a number of random hops. After this step, the message is broadcast over the whole network using diffusion. With this combination of random path selection and diffusion, the **Dandelion** protocol provides near-optimal anonymity guarantees.

This protocol can be visualized using the following diagram:

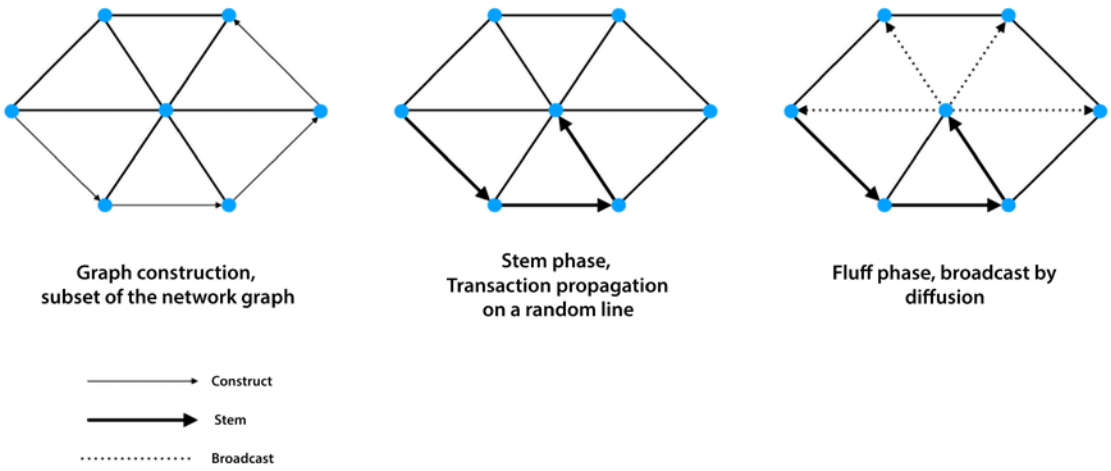


*Figure 18.2: Dandelion protocol routing*

In the preceding diagram, first, graph construction is performed, which is shown on the left-hand side. Then, in the middle, we have the transaction propagation along a random line of nodes. Finally, the broadcast, on the right-hand side, is shown. More information on Dandelion is available in the following paper: `https://arxiv.org/pdf/1701.04439.pdf`.

## Confidential transactions

Confidential transactions make use of **Pedersen commitments** in order to provide confidentiality. Commitment schemes allow a user to commit to some value while keeping it secret with the capability of revealing it later. Commitment schemes can be constructed simply by using cryptographic hash functions. Two properties that need to be satisfied in order to design a commitment scheme are binding and hiding.

**Binding** makes sure that the committer is unable to change the chosen value once committed, whereas the **hiding** property ensures that any adversary is unable to find the original value to which the committer made a commitment.

A Pedersen commitment is a type of information-hiding, computationally binding commitment scheme, under a discrete logarithm assumption. Pedersen commitments allow for additional operations and preserve commutative property on the commitments, which makes them particularly useful for providing confidentiality in **Bitcoin** transactions. In other words, they support partial homomorphic encryption of values, meaning that using commitment schemes allows us to hide payment values in a Bitcoin transaction. This concept is already implemented in the *Elements Project*: `https://elementsproject.org/features/confidential-transactions`.

## MimbleWimble

The **MimbleWimble** scheme was proposed anonymously on the Bitcoin IRC channel and since then has gained a lot of popularity. MimbleWimble extends the idea of confidential transactions and Coinjoin, which allows the aggregation of transactions without requiring any interactivity. However, it does not support the use of the Bitcoin scripting language, along with various other features of the standard Bitcoin protocol. This makes it incompatible with the existing Bitcoin protocol. Therefore, it can either be implemented as a sidechain to Bitcoin or on its own as an alternative cryptocurrency. It was however implemented in Litecoin in May 2022.

This scheme can address both privacy and scalability issues at once. The blocks created using the MimbleWimble technique do not contain transactions as in traditional **Bitcoin** blockchains; instead, these blocks are composed of three lists: an input list, an output list, and something called *excesses*, which are lists of signatures and differences between outputs and inputs. The input list basically references the old outputs, and the output list contains confidential transaction outputs.

The blocks created using the MimbleWimble scheme are verifiable by nodes by using signatures, inputs, and outputs to ensure the legitimacy of the block. In contrast to Bitcoin, MimbleWimble transaction outputs only contain `pubkeys`, and the difference between old and new outputs is signed by all participants involved in the transactions.

## Zkledger

This distributed ledger provides transaction privacy, including confidentiality of transaction amount and sender and receiver addresses. It utilizes NIZKs and Sigma protocols to achieve this. However, this protocol is not scalable because the size of transactions grows linearly with the number of parties involved.

## Attribute-based encryption

The **attribute-based encryption** (**ABE**) is a type of public key cryptography that provides confidentiality and access control simultaneously. The key idea behind this scheme is that the private key of a user and its ciphertext are dependent on the user's attributes such as their location, time zone, or their role in the organization. This means that decryption is only possible when not only the private key is available but also the matching attributes are present.

## Anonymous signatures

Anonymous signatures are types of digital signatures where the signatures do not reveal the identity of the signer. There are primarily two schemes available for anonymous signatures: **group signatures** and **ring signatures**.

Group signatures allow a set of signers to form a group managed by a group manager. Each member of the group is issued with a group signing key by the group manager. This group signing key allows each member of the group to anonymously sign messages on behalf of the group. The group manager is able to figure out who is the signer of the message, whereas external entities cannot. While group signatures can work well, one of the limitations of this scheme is that the group manager is able to identify the users. This issue was addressed with ring signatures.

**Ring signatures** allow a set of signers to form a group (a ring) of members. Each member in this group is able to sign messages on behalf of the ring. Unlike group signatures, there is no group manager in ring signatures, so no one is able to identify the signers.

While all these techniques are useful and provide many desirable properties of anonymity, there is also a problem with using this technology maliciously. Imagine if anonymity is misused by criminals involved in money laundering or selling illegal drugs by using anonymous cryptocurrency. It would be almost impossible to trace illegal activities back to the originator if absolute anonymity is achieved. Imagine a **Silk Road marketplace** (`https://en.wikipedia.org/wiki/Silk_Road_(marketplace)`) variant using a fully anonymous and confidential cryptocurrency. How could that be traced and stopped?

So far we have discussed generic cryptographic techniques and some examples thereof to enable privacy in blockchain. However, we haven't touched on the important concept of private smart contracts. Programmable blockchains, starting from Ethereum and now chains like Solana and Polkadot, all support smart contracts. It is desirable to achieve smart contract-level privacy too, instead of only standard transaction-level privacy. Think of an ERC-20 contract where all operations like transfers, balance checking, and others are performed with full privacy. There are several proposals to do that as well, some of which we'll discuss next.

## Zether

Zether allows a private transaction mechanism that supports confidentiality and anonymity. It is implemented on the Ethereum blockchain but, in theory, can be implemented on any programmable chain. Zether is implemented as a smart contract where ether is deposited and turned into a Zether token.

Other proposals include Zexe (`https://eprint.iacr.org/2018/962.pdf`), PPChain (`https://ieeexplore.ieee.org/document/9199417`), Hawk (`https://ieeexplore.ieee.org/document/7546538`), and ZKledger (`https://dci.mit.edu/zkledger`).

## Privacy using Layer 2 protocols

Privacy using Layer 2 and state channels is also possible, simply because all transactions are run off-chain, and the main blockchain does not see the transactions at all except for the final state output and minimal transaction data (representing transactions), which ensures anonymity and confidentiality. However, note that usually privacy is not the main purpose of Layer 2 protocols, instead they are more concerned with provided scalability. Nevertheless, Layer 2 can be used to provide anonymity and confidentiality if zero-knowledge is used not only as proof of correct computation (integrity) but also as a confidentiality measure. This is achieved in Aztec (`https://aztec.network`).

## Privacy managers

Privacy managers provide a mechanism that provides confidentiality of transactions. These are off-chain components that substitute the transaction payload with a hash index, in such a way that only those participants who are party to the transaction are able to find the corresponding encrypted payload represented by the hash index. Other parties who are not privy to the transaction will simply ignore the hash. This concept was discussed in *Chapter 16*, *Enterprise Blockchain*.

Other than scalability and privacy, there are several other general security aspects that need to be addressed in blockchain. We'll describe these general security topics in the next chapter, *Chapter 19*, *Blockchain Security*.

There are two approaches when it comes to scalability and privacy solutions. First is the development of domain-specific languages, which can be used to write programs that can be verified by a verifier on Layer 1. Secondly are ZK virtual machines, which allow computations (programs) to be verified on Layer 2. DSLs require developers to learn a new language whereas ZK VMs are compatible with mainstream tools and languages already in use. A key advantage of DSLs is that the generated ZK proof or circuit is agnostic to the underlying chain, which makes it easy to use it on any chain, as long as we write a verifier for that chain. ZK VMs however have some risk of vendor lock-in.

# Privacy using zero-knowledge

When we think about privacy, the first thing that comes to mind is cryptography and we can think of several ways to achieve privacy, which we describe next:

1. We can possibly encrypt transactions so that data is confidential, but this approach is impractical for two reasons. First, public verification of transactions is not possible as the values are hidden; secondly, even if some encryption scheme is utilized, key management becomes a big issue due to the sheer number of keys required.

2. The second approach is commitment schemes, which is quite a promising solution. We can replace the amount/value of the transaction with a commitment to the value, then we commit to the value and post it on the blockchain. Again, the issue is that as the value is hidden, the transaction is not publicly verifiable; also, commitment to a negative value is possible.
3. The solution is to attach ZFPs with the commitment, which proves two things: range commitment, which ensures that the number is positive and is in a valid range, and value commitment, which hides the actual transaction value. Range commitments are harder to do whereas value commitment is comparatively straightforward, as it translates as a simple linear equation, which is easier to handle.

Let's now explore cryptographic commitment, an essential cryptographic primitive to build zero-knowledge protocols on blockchain.

## Cryptographic Commitments

A cryptographic commitment scheme is a two-phased protocol composed of two algorithms, a `commit` algorithm, and a `verify` algorithm:

- `Commit(m, r) -> c`
- `Verify(m, c, r) -> accept or reject`

In the first phase, called the commit phase, the sender sends the receiver a commitment string `c`. Alice runs the `commit` algorithm to commit to a message `m` using secret randomness `r` and produces a commitment string `c`. This commitment string `c` is sent to Bob.

In the second phase, called the open phase, the receiver opens the commitment to verify that the sender indeed committed to the message `m` and did not cheat.

This is checked using the verify algorithm. This algorithm takes three values, message `m`, commitment `c`, and randomness `r`, and based on the output the verifier either accepts or rejects the commitment. Once the commitment is open anyone can publicly verify that the commitment is opened correctly.

We can think of cryptographic commitments as a digital analog or cryptographic equivalent of sealed envelopes. There are two security properties of cryptographic commitments, namely hiding and binding:

- The hiding property ensures that the committed message is not revealed before the open phase. In other words, commitment `c` reveals nothing about the committed message.
- The binding property ensures that once the value is committed, it cannot be changed later on. In other words, once the committer has committed to a value, the committer must not be able to open the commitment with a different message `m`, which the verify algorithm also accepts.

There are two types of commitment schemes based on the strength of the security properties:

- Standard commitment schemes, which protect against a PPT receiver and an unbounded powerful sender. Such schemes are computationally hiding and information-theoretically binding.
- Perfect commitment schemes, which protect against and PPT sender and an unbounded powerful receiver. Such schemes are information theoretically hiding and computationally binding.

Commitment schemes that are both information theoretically hiding and binding do not exist.

A **Probabilistic Polynomial Time** (**PPT**) adversary has access to a randomized polynomial time algorithm to break the security of the system. We can think of it as a computationally bound entity that can only break the security of the system if it has access to a PPT algorithm. Polynomial time algorithms are those algorithms that run in a "reasonable" amount of time or take a reasonable time to compute. More formally, a polynomial time algorithm is an algorithm that gets an input of size $n$; it would be considered polynomial time if it runs in $O(n^c)$ time where $c$ is a constant. A PPT algorithm is a polynomial time algorithm that is randomized, meaning that it has access to a source of randomness, i.e., allowed to flip coins.

There are several models of commitment schemes including **hash-based commitment schemes**, **Pedersen commitments**, and **polynomial commitment schemes.** A type of commitment scheme called Pedersen commitment is very useful due to its homomorphic properties. Let's first see how the Pedersen commitment is built and then we'll see how its homomorphic property works.

The scheme works in three phases, public setup, commit, and open.

Setup phase:

1. Let $\mathbb{G}$ be a finite cyclic group of prime order
2. Get two elements $g, h \in \mathbb{G}$ , where $g$ is a known fixed group element called *generator* and $h$ is a uniformly random group element from $\mathbb{G}$
3. Let $q$ be the order of $\mathbb{G}$ which is exponentialy large

Commit phase:

1. The committer commits to a number $m \in \{0, \ldots . q - 1\}$
2. The committer picks a random integer $z \in \{0, \ldots q - 1\}$
3. Sends the commitment $c$ to the receiver, which is $c \leftarrow g^m . h^z$

Open phase:

1. To open the commitment $c$, the committer sends $(m, z)$
2. The receiver verifies that $c = g^m . h^z$

The Pedersen commitment is perfectly hiding and computationally binding. Pedersen commitments are additively homomorphic. Suppose we have two commitment strings, $c_1$ and $c_2$, which are commitments to messages $m_1$ and $m_2$ respectively. If we multiply these two commitments the result is a new commitment $c_3$ to the sum $m_3$ of $m_1$ and $m_2$. This means that without knowing the original $m_1$ and $m_2$ (the originally committed messages) the receiver can construct a commitment to the sum of the committed values, i.e., a commitment $c_3$ to $m_3$, where $m_3 = m_1+m_2$ and $c_3$ can be opened to $m_3$ by using the opening information for $c_3$, which is $(m_1+m_2)$.

Pedersen commitments are commonly used with some modifications in several privacy-preserving blockchains, including Monero and Zcash. Note that Pedersen commitments alone cannot create a complete privacy-preserving solution, but they serve as a component in a larger solution that includes ZKPs, which we discuss next.

## Zero-knowledge proofs

The first ZK proof was invented by Goldwasser, Micali, and Rackoff in 1985. Succinct transparent arguments from PCP were introduced by Kilian and Micali in 1992 and 1994 respectively. These proofs are the first SNARKs, but they were impractical due to impractical prover time; nevertheless the proofs were short in size and fast to verify. For a long time, ZKPs were considered impractical but in 2013 a breakthrough was made that introduced linear prover time with a constant size proof. This was later improved in 2016. Both papers are available here (GGPR13 – `https://eprint.iacr.org/2012/215` and Groth16 – `https://eprint.iacr.org/2016/260`).

In 2013, Pinocchio, the first practical ZK SNARK, was introduced. Also, tinyRAM was introduced in 2013, which is the first ZK VM. In 2018, the first ZK rollup from Barry Whitehat, the Circom language form `iden3`, and the first ZKSTARK from Starkware were introduced. In 2019, PLONK, a major milestone, was achieved, which improved the usability of SNARKs. Its setup is universal and updateable, which means that the setup needs to run only once and then can be used for any program instead of running a separate setup for each new program, as was the case in the earlier schemes, e.g., Zcash. Secondly PLONK uses polynomial commitment, which is swappable with any other commitment scheme to achieve different improvements. This means that the proving scheme and the commitment scheme are separate, and a different more efficient commitment scheme can be used. Plonk uses KGZ10 (Kate) but it can be swapped with FRI or some other scheme.

The acronym **FRI** stands for **Fast Reed-Solomon IOP of Proximity**, and **IOP** stands for **Interactive Oracle Proof**. The FRI protocol ensures that a committed polynomial has a bounded degree.

FRI would allow it to become transparent and post-quantum secure (resistant), while keeping the proving scheme the same, i.e., PLONK. Earlier proving schemes like Marlin and Sonic were more complex and costly. Then in 2020, Cairo ZKVM and Halo2 were developed. 2021 saw Circom 2.0 and Mina development. In 2022, a lot of projects were introduced, including Plonky2, which is the fastest ZKSNARK so far, along with many other projects, such as ZKEVMs, including Scroll and zkSync. There is no sign of this development stopping or even reaching a plateau anytime soon; more and more research and development are being carried out by enthusiastic teams, and this trend is only expected to grow further with more innovation on the horizon. As it is now an established belief that ZK very nicely complements blockchains, the future of blockchain is going to be heavily oriented around zero-knowledge.

Note that *proof* and *argument* are two terms that are used sometimes interchangeably. In S*ARKs, **AR** stands for **arguments**, indicating that these are not proofs. Zero-knowledge proofs are however proofs, and not arguments. It's important to understand this difference. The difference between an argument and a proof is to do with how strong the soundness property in the protocol is. In the case of arguments, the soundness property is secure against a polynomially bounded adversary (prover) whereas proofs are secure against a computationally unbounded adversary (prover). In other words, proofs are information theoretically secure whereas arguments are computationally secure.

The first cryptocurrency to successfully implement ZKPs (specifically ZK-SNARKs - **Succinct Non-Interactive Argument of Knowledge**) to ensure privacy on the blockchain is Zcash. The same idea can be implemented in **Ethereum** using smart contracts and other blockchains also. The original research paper is available at `https://eprint.iacr.org/2013/879.pdf`. Another excellent paper is available here: `http://chriseth.github.io/notes/articles/zksnarks/zksnarks.pdf`.

There is another type of ZKP called **zero-knowledge Succinct Transparent Argument of Knowledge** (**zk-STARK**), which is an improvement on zk-SNARKs in the sense that zk-STARKs consume a lot less bandwidth and storage. Also, they do not require the initial, somewhat controversial, trusted setup that is required for zk-SNARKs. Fundamentally they are SNARKs but without a trusted setup. Moreover, zk-STARKs are much quicker as they do not make use of elliptic curves or rely on hashes. The original research paper for zk-STARKs is available here: `https://eprint.iacr.org/2018/046.pdf`.

**Bulletproofs** are non-interactive zero-knowledge short proofs. They require no trusted setup. This scheme allows a prover to prove that an encrypted number is within a range of numbers without revealing any other information about the number. More information on Bulletproofs is available here: `https://electroneropulse.org/public/doc/Bulletproof%20RingCT.pdf`.

A quick comparison of the techniques mentioned above is shown below:

| Type/Property | Proof size | Prover time | Verification time | Trusted setup required |
|---|---|---|---|---|
| SNARKs | Small<br>288 bytes | Medium<br>~2.3 seconds | Small<br>~10 milliseconds | Yes |
| STARKs | Large<br>~40 – 50 KB | Small<br>~1.6 seconds | Medium<br>~16 seconds | No |
| Bulletproof | Medium<br><1 KB | Large<br>~30 seconds | Large<br>~1100 seconds | No |

We covered ZKPs in *Chapter 21* and *Chapter 4*. Here we'll focus more on non-interactive ZKPs and cover SNARKs in more detail.

Generally, there are two classes of zero-knowledge arguments, sigma protocols and SNARKs.

Sigma protocols are an efficient way to create ZKPs. These are three-round protocols with a proof, challenge, and response phase with a verifier, who is able to throw a public coin (randomness), as shown in *Figure 18.3* below:

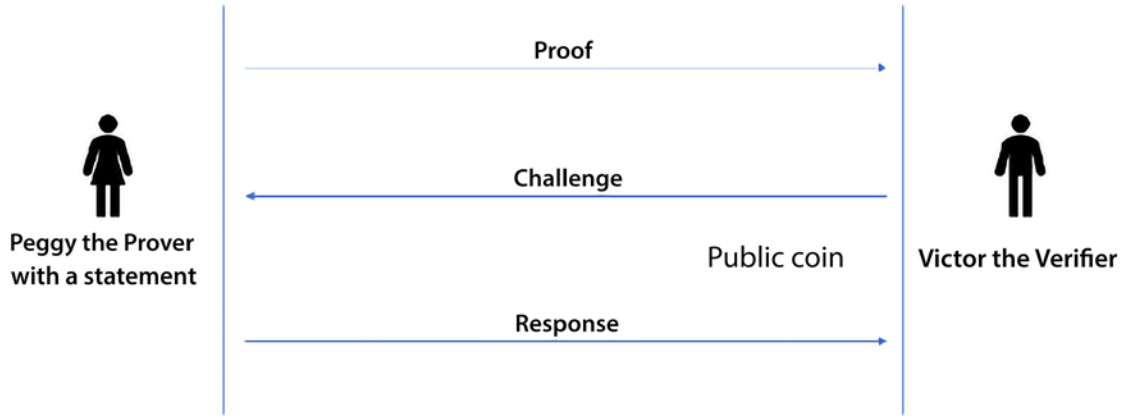


*Figure 18.3: Sigma protocols*

As SNARKs are the most used and promising technology to implement privacy in blockchains, we'll focus on exploring SNARKs more.

As we saw in *Chapter 4*, ZKPs traditionally have large data structures and require multiple challenge-response interactions between a prover and verifier for the prover to prove some assertion. Interactive protocols can be useful in limited scenarios; however, NIZKs are more suitable for blockchain-based scenarios where the prover can post the proof online/on-chain and verifier(s) can verify independently (asynchronously). This is shown in *Figure 18.4* below:

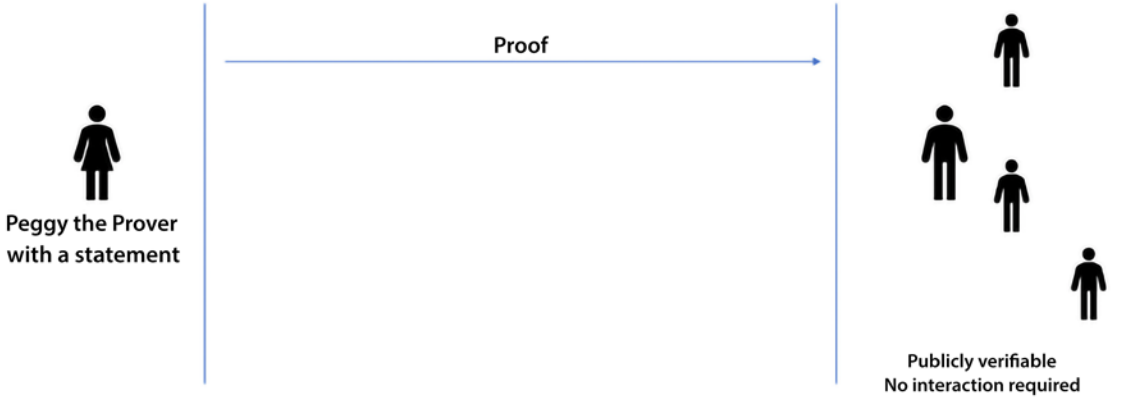


*Figure 18.4: NIZK*

For achieving non-interactivity several techniques can be employed, including a **common reference string model** where participants share a random string. It is crucial for NIZK proofs; without CRS, NIZK cannot exist. Practically speaking CRS is a set of elliptic curve points of a specific form.

A typical visual representation of a CRS scheme is shown in *Figure 18.5*:

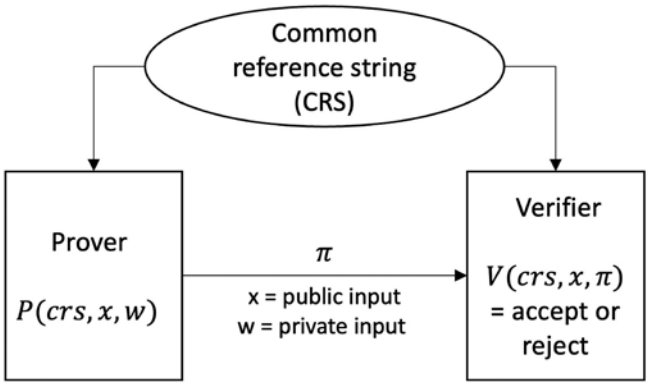


*Figure 18.5: CRS-based proving scheme*

Trusted setups are usually called setup ceremonies, where the proving key for the prover and the verification key for the verifier are generated. Here some "toxic waste" is also produced, which needs to be destroyed. Here *toxic waste* simply means some secret randomness that needs to be destroyed privately; otherwise, the security of the system is compromised.

There are four methods that can be used to generate reference strings, namely trusted, subverted, transparent, and MPC:

- A **trusted model** simply means that there is just a single trusted party generating the reference string, which clearly is not acceptable in practice because if that single party turns malicious, there is no way to stop it.
- **The subverted approach** says that even if the prover has generated the SRS (i.e., knows the trapdoor, or in other words, fully subverted the setup process) even then it won't be able to prove a false proof. This of course again is not practical, as it has been proven as an impossibility result that ZK cannot be achieved using this method.
- In **transparent setups**, instead of SRS, a **uniform reference string** (**URS**) is randomly generated and used, which means that no toxic waste (a trap door, i.e., keys and randomness, etc.) is produced. Note that the polynomial commitment scheme used in the protocol entirely dictates the transparency of the SNARK; if the PCS uses URS, then the SNARK protocol also uses URS. This option enables trustless setup.
- **The MPC-based approach** allows multiple parties to jointly generate a structured reference string in a decentralized manner without compromising security. As long as even a single participant has deleted their part of the secret, the entire scheme can be considered secure. **Multiparty computation** (**MPC**) is a method of allowing multiple parties to cooperatively compute a function over their private inputs, without revealing those inputs to each other. MPC enables multiple parties to collaborate and execute a computation while keeping their inputs and intermediate results private.

Trusted setup has been performed in Zcash, Aztec, and several other protocols. The caveat here is that if the SRS is not universal then this setup ceremony needs to be run every time the circuit requirements change, e.g., a new type of transaction or changes in the existing transaction structure. This is why Zcash had to run the ceremony again in 2018 (the Sapling ceremony), after the first time in 2016 (the Sprout ceremony).

Zcash has released Orchard shielded payment protocol, which uses the Halo 2 proving system and removes the need for a structured reference string and thus the trusted setup ceremonies. This means that future circuit upgrades can be done without a trusted setup. More information on this is available here: `https://zips.z.cash/zip-0224`.

We can also divide SNARKs into three types based on the setup requirements. There are three main types of setups:

- **Trusted non-universal setup** (specific purpose – trusted setup per circuit)-based SNARKs where they are built for a single circuit. They have a large overhead of a large common reference string. Here the setup procedure generates some random data that must be kept secret; otherwise, the prover will be able to prove false statements. Once the setup is done, this data is destroyed. They are used first in Zcash; it's OK even with the limitations because the setup doesn't need updating, as there is only a single type of circuit, i.e., a transaction transfer. The setup must be done again if the circuit type changes.
- **A trusted universal setup** where the SNARK only requires a trusted setup once. They have a **structured reference string** (**SRS**), which is updatable and smaller than CRS. Here the reference string has some structure based on secret randomness. This is also a trusted setup but is universal and updatable because the secret material (randomness, trapdoor, etc.) in this setup is independent of the circuit. This can be thought of as a two-step process where an initial one-time procedure runs secretly to generate an updateable setup string and any secrets are destroyed. However, now this setup string is updatable for any new circuit and can preprocess any new circuits without relying on any secret data.
- The third type is **transparent SNARKs** or STARKs, which require no trusted setup as they don't use any secret data.

As SNARKs produce small proofs (succinct) that are quick to verify, these proofs can be used for not only privacy but also for scalability, as we saw in *Chapter 17*, *Scalability*, because they are short in size, and verifiers only have to process and verify a small piece of data instead of all transactions. These proofs, while efficient, have two key issues. First it takes a long time to generate proofs and secondly SNARKs require SRSs, which means trusting third parties.

Let's now see how ZK-SNARKs are built.

## Building ZK-SNARKs

Generally speaking, generating a SNARK is a two-step process. The first step, which involves the "**frontend**" of the system, is to convert a program into an equivalent model that can be checked probabilistically.

This means transforming the program to be proven into an arithmetic circuit, i.e., generally into a circuit satisfiability problem. The arithmetic circuit is further represented by an arithmetic constraint system, which can be R1CS – the **Rank 1 constraint system** or **Hadamard Product Relation** (**HPR**). At a high level the constraint system encodes the circuits as matrices. The first step can be seen as a process where the program is transformed into an equivalent circuit, which basically allows us to make checking for the proof by a verifier non-interactive. The program code is transformed into an arithmetic circuit, which is an algebraic circuit that represents the program code. Practically the arithmetic circuit is described as matrices and relevant operations in linear algebra. The prover and verifier run the SNARK for circuit satisfiability. Here usually DSLs like Circom and Zokrates are used to write the circuit. Circom is a lower-level language where the entire program is written almost gate by gate, Zokrates is a bit higher level than that and allows us to create programs in a more familiar Python-style code. Cairo and Noir are two other languages that can be used to write circuits to represent the program. This approach of using a DSL to write programs is much easier than writing a compiler to take a high-level program written in Rust or C and convert that directly into an equivalent circuit. Such an approach is not much practical; therefore, specific DSLs are used for writing circuits.

Any program that the prover wants to prove is first converted into an arithmetic circuit, which is also called *flattening*. An arithmetic circuit is a construct that represents a mathematical function as a **directed acyclic graph** (**DAG**) consisting of gates, inputs, wires, and outputs. Gates are vertices and wires are edges. Each gate in the circuit performs a specific arithmetic operation (such as addition or multiplication) on its inputs and produces an output, and the outputs of the gates are used as inputs to other gates until the final output is produced. For carrying values, wires (edges) are used. The inputs' nodes are labeled with variables and constants.

To build ZK-SNARKs, arithmetic circuits are used to represent the computation that the prover wants to prove. More precisely the prover tries to convince the verifier that it has a solution to the arithmetic circuit.

Such a circuit is shown in *Figure 18.6* below, which computes the expression (*a*+*b*) x *b* x *c*:

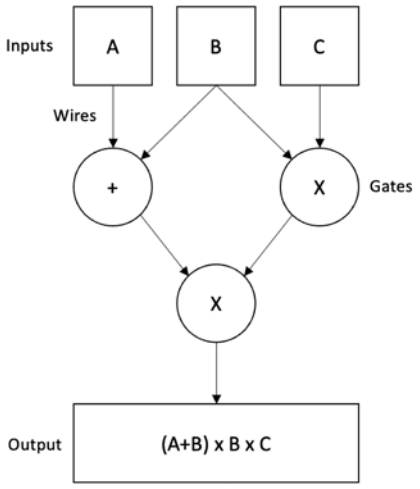


*Figure 18.6: An arithmetic circuit*

Polynomials are extensively used to represent many problems in cryptography. Circuits are also defined over a prime finite field as multivariate polynomials, and the circuit structure provides a computational model to evaluate the polynomial.

The key feature of ZKPs based on arithmetic circuits is that they enable a prover to produce a proof that shows that the prover knows an input value for the circuit, which produces a predefined output while keeping all or some inputs secret. These proof systems ensure that a prover is not able to generate a valid proof without the knowledge of correct inputs for the circuit and that the private inputs are obfuscated, so that no information is leaked about private inputs.

We saw earlier, what an argument and a proof are. Let's now define the ZK argument system in light of what we have learned about circuits. In an argument system we have two parties, a verifier and a prover. Imagine we have a circuit that takes two inputs, `x` and `w`. `x` is the public input and `w` is the secret witness that the prover is trying to prove knowledge of. The prover's aim is to convince the verifier that there is a witness `w`, which when processed by the circuit along with the public input `x` produces the desired output. The verifier doesn't know what `w` is and only knows `x`, the public input. The verifier and prover engage in several rounds of exchanging messages, and at the end of the protocol the prover either rejects or accepts the proof. In other words, the verifier is either convinced that the verifier indeed knows x, or not. A non-interactive zero-knowledge argument system has four properties: completeness, the argument of knowledge, zero-knowledge, and succinctness:

- **Completeness** means that an honest prover is always able to convince an honest verifier that the proof is correct.
- An **argument of knowledge** means that if the verifier has accepted the proof, then the prover undeniably knows a secret piece of information, called a **witness**, that satisfies the circuit. In other words, the valid proof can only be generated by the prover who indeed knows the private input. More precisely this means that the prover knows the witness if it can be extracted from the prover. The extraction means that an efficient procedure exists that can extract the secrets from a prover. An argument system is computationally sound, instead of information theoretically sound, where the assumption is that it is impossible to prove a false statement to the prover. In an argument system we say it is infeasible to fool the prover instead of impossible. If we assume impossible, then it's a proof instead of argument.
- **Zero-knowledge** means that the proof produced by the provers does not reveal anything at all about the witness. In other words, it doesn't reveal anything apart from whether the statement asserted is true or false. Formally this property means that if the verifier is able to regenerate the proof on its own it would learn nothing new; the proof would still not reveal anything about the witness. We can say that a ZK argument system is zero-knowledge for a circuit if there is an efficient procedure, called a **simulator**, that can generate public parameters and the proof, which are indistinguishable from the real public parameters generated by the setup procedure and the proof generated by the real prover by only using public input x, i.e., without knowing the witness.
- **Succinctness** means that the proof produced is small in size and is quickly verifiable by the verifier. The speedup in verifier time is in fact achieved because of the setup phase, where public parameters are generated.

> We can think of the setup phase as the circuit being "preprocessed" in a way that the verifier doesn't have to read the entire circuit. The prover of course has to read it fully, but the verifier can avoid it by relying on essentially a "short" version of the circuit produced, due to the pre-processing performed in the setup phase.

After the circuit is generated, the public parameters can be generated by the setup procedure, which takes the arithmetic circuit and produces the required parameters. These public parameters are used by the prover along with the public input $x$ and secret witness $w$ to produce the proof (usually denoted by pi $\pi$). The verifier also uses these public parameters along with the public input $x$ to verify the proof sent by the prover.

SNARKs rely on a trusted setup where a common reference string is generated to make the proof sizes smaller. We have several types of reference strings, including CRS, SRS, and URS. The SRS is essentially a proving key and is usually as big as the circuit. There is a trapdoor function in this SRS, and if the prover discovers it, they can easily create false statements that they can prove to the verifier as true. This trapdoor is in fact known to the entity that generates this SRS, and this entity needs to be trusted by the system. The expectation is that this party will destroy this "toxic waste" after the SRS is set up. This is of course not ideal, and a lot of research has been done to minimize the trust requirements. One key example is creating SRS through the MPC protocol. There is however no requirement for an SRS setup in STARKs. SRS for simpler circuits is manageable, but complex circuits with possibly billions and trillions of gates can become a problem to manage in terms of storage and distribution to provers.

In the second step, involving the **backend** of the system, a proving scheme is applied for circuit satisfiability. Some examples of the proving schemes include Groth16, GM17, Marlin, PLONK, Libra, Supersonic, Hyrax, and many others. The key requirement behind the development of the backend is to create a SNARK that both the prover and verifier can run to satisfy the circuit. The proving scheme is expected to have fast verifier verification times, be succinct (small in size – a few bytes or KBs), and be at least linear (in terms of algorithmic complexity) to the program size for prover times.

We can visualize the frontend, setup, and backend as a ZK-SNARK system in *Figure 18.7* below:

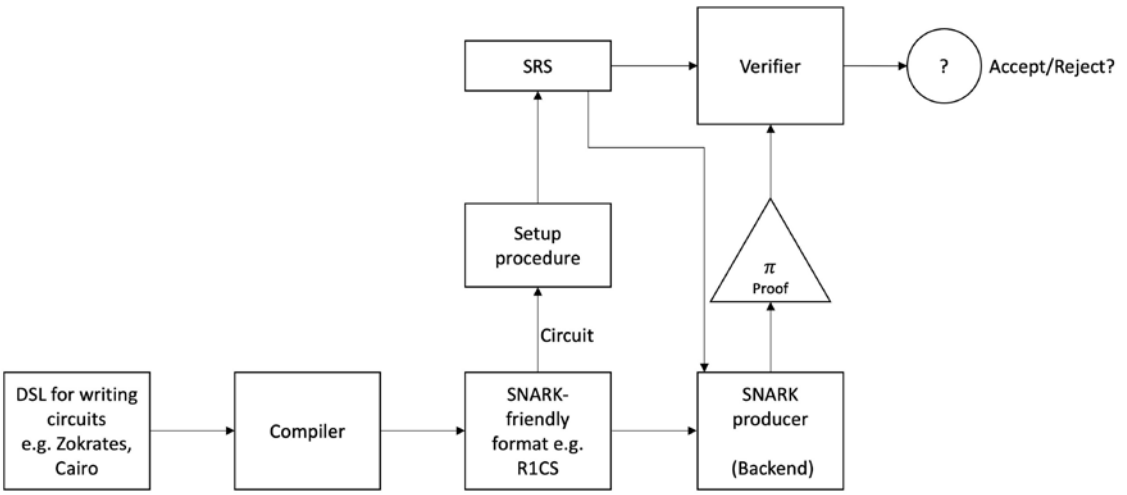


*Figure 18.7: A typical ZK-SNARK system*

The design of the second step, i.e., the backend or proving scheme, generally includes three steps:

- Create (use) a **Polynomial Interactive Oracle Proof** (**PIOP**) for circuit satisfiability. In other words, create a proof scheme that proves that the statement (program) for which the circuit has been built is true.
- Create (use) a polynomial commitment scheme to build a succinct interactive argument.
- Use the Fiat-Shamir heuristic to transform this interactive argument into a non-interactive protocol. This is achieved by substituting the verifier's randomness with a random oracle, which is instantiated by a cryptographic hash function. The prover is unable to predict the outcome of the hash function, thereby providing the randomness necessary to achieve the soundness property of the ZKP.

Combining the first two steps in the "backend" produces a SNARK, i.e., PIOP + PCS = SNARK.

We have used the term *IOP* quite a bit so far. Let's see what this is. **IOP** stands for **Interactive Oracle Proof**. It is an interactive proof in the sense that the verifier doesn't have to read the entire message from the prover; instead, the verifier has access to an oracle for the prover's message and can query it probabilistically. Practically this means that the verifier queries some evaluation points instead of reading the entire message. Here the prover has a public input and a witness, whereas the verifier has only the public input. Prover messages are some arbitrary bit strings ($d$ i.e. polynomial encodings), whereas verifier messages are some randomness (i.e. queries on evaluation points). The prover and verifier engage in several rounds of these messages and eventually, the verifier generates queries to query the oracle at particular locations (evaluation points).

Once the result of these queries is generated the verifier decides whether to accept or reject the proof. To ensure that the messages are indeed polynomials a test called a "low-degree test" is performed, which checks the degree of the polynomial. However, this produces large-size proofs and requires a number of rounds to operate, which is quite costly. However, if we can assume that messages sent from the prover to the verifier are polynomials over a finite field, then there is no need for a low-degree test, and this will result in a low cost. This is exactly what polynomial IOPs do when prover messages are polynomials and because of this assumption, there is no low-degree test, which reduces cost and improves efficiency. IOPs can be categorized into three classes based on how they've evolved over time:

- **Interactive proofs**, which are a standard category where a single prover and a verifier engage in a protocol to prove the assertion of a statement by the prover. Some examples of SNARKs in this category are Libra and Hyrax. The prover time is lowest in this category, but the proof size is big and the verifier time is high.
- **Multi-prover interactive** proofs, where instead of a single prover, multiple provers exist. Some key examples in the category are Spartan and Brakedown.
- **Constant round PIOPs** are IOPs with constant round complexity. Key examples are Marlin and PLONK. The prover time is high in this category; however, the proof size and verifier time are fast.

After a PIOP is created, it's time to use an existing or create a new **polynomial commitment scheme** (**PCS**) to build a succinct interactive argument. Polynomial commitment schemes allow a committer (prover) to commit to a polynomial by sending a short string to the verifier.

Later the verifier can ask the prover to evaluate a single point in the polynomial. The prover responds with the result and a proof, which the verifier evaluates. The aim here is that the prover must not be able to produce a convincing proof of a wrong evaluation of the point on the polynomial. Secondly commitment and the proof are easy to generate and small, and the proof is easy and quick to verify. Polynomial commitment schemes can be classified into four categories according to the cryptographic assumptions they are based on:

- **IOP** with a **collision-resistant cryptographic hash function**, which doesn't require a setup (i.e., is transparent) and is post-quantum resistant. Most common examples of this scheme include FRI (`https://drops.dagstuhl.de/opus/volltexte/2018/9018/pdf/LIPIcs-ICALP-2018-14.pdf`) and Ligero commitments.
- **ECDLP-based**, which uses elliptic curves and is transparent but not post-quantum resistant. Bulletproofs (`https://eprint.iacr.org/2017/1066.pdf`) are a common example of such a type of commitment scheme.
- **Pairing group-based**, which requires a trusted setup and is also not post-quantum resistant. A common example is the KZG10 (`https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf`) commitment scheme. Another is Dory.
- **Groups of unknown order-based**, which are transparent if using class groups but not post-quantum resistant. One major drawback is very slow prover times, which is due to the use of class groups. A common example of such schemes is DARK (`https://eprint.iacr.org/2019/1229.pdf`).

We can visualize a generic construction pipeline of SNARKs based on what we have discussed so far in *Figure 18.8*. We can also view this as a process of how we go from a program to SNARK:

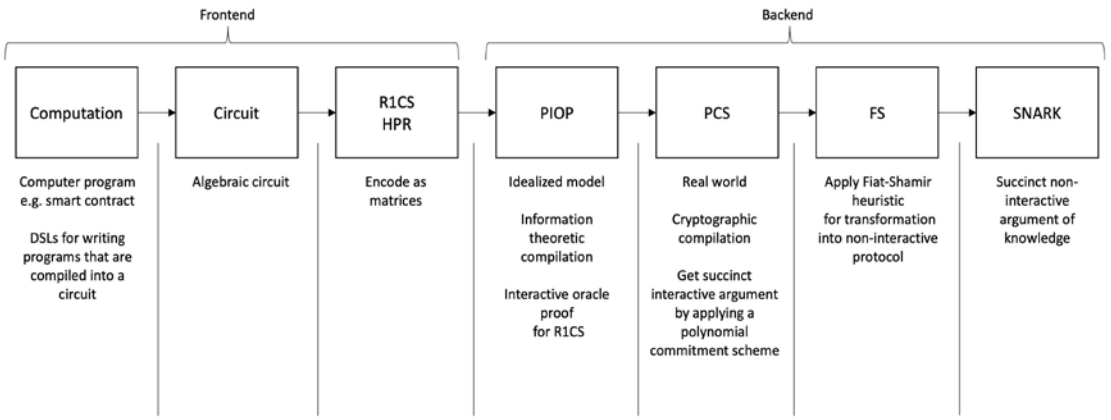


*Figure 18.8: Generic construction pipeline of a SNARK*

In summary, the construction pipeline starts with the arithmetization of a computation where techniques like R1CS and HPR are used. The next step is the compilation of this arithmetic representation into an information theoretical protocol, where idealized models such as **polynomial IOPs** (**PIOPs**), PCPs, and linear PCPs based on oracles are used.

Once we are happy with the information theoretic construction, to make it practical and real, we replace the information theoretic model with the cryptographic constructs called polynomial commitments. Here we essentially compile IOP to SNARK. A usual candidate for PCS is the Kate commitment scheme (`https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf`). This is where finally the SNARK is produced. We can think of it as combining PIOP with PCS to yield a SNARK scheme.

A polynomial commitment scheme allows a committer to commit to a polynomial. The hiding and binding properties apply to polynomial commitment schemes just as other commitment schemes. They are also composed of commit and open phases like other commitment schemes. However, there is an additional property that we can call "selective opening." This property means that the committer is able to open certain evaluations of the commitment without revealing the entire polynomial. In other words, the PCS allows commitment to a polynomial with a short string, which can be used by the receiver to verify the claimed evaluations of the committed polynomial. This means that the prover can prove that it possesses a polynomial, which satisfies some required properties without disclosing the polynomial.

Now let's see what the Kate PCS is and how it works. The scheme can be divided into three phases, setup, commit, and open and verify, just like other commitment schemes.

## Setup

This is the one-time trusted setup to generate the SRS:

- Let $e$ be a bilinear map pairing groups $\mathbb{G}, \mathbb{G}_t$ of prime order $p$.
- Let $g \in \mathbb{G}$ be a generator. $\mathbb{G}$ is a pairing-friendly elliptic curve group.
- Let $l$ be the maximum degree, i.e., the upper bound on the degree of the polynomials we want to be able to commit to.
- Pick a random integer $\tau$ from the prime field, i.e., $\tau \in \mathbb{F}_p$ , i.e., $\{1, \ldots, p-1\}$.
- Compute SRS as $(g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^l})$. This SRS consists of encodings in $\mathbb{G}$ of all powers of $\tau$.
- Note that $\tau$ is the "toxic waste," i.e., the secret parameter (or trapdoor) of the setup ceremony, which must be destroyed once the SRS is generated.

## Commit

In this phase, commitment to a polynomial $\phi$ over $\mathbb{F}_p$ is made:

- A polynomial $\phi(x) = \Sigma_{i=0}^{l} \phi_i x^i$.
- The compute commitment $c = g^{\phi(\tau)}$. Here even if the $\tau$ is destroyed the committer can still compute this using the SRS and additive homomorphism. Recall that our SRS is $(g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^l})$. If $\phi(x) = \Sigma_{i=0}^{l} \phi_i x^i$ then $g^{\phi(\tau)} = \Pi_{i=0}^{l} \left(g^{\tau^i}\right)^{\phi_i}$, meaning that given the values $g^{\tau^i}$ for all $i = 0, \ldots, l$ without knowledge of $\tau$, commitment $c$ can be computed.

In simpler terms this means that we evaluate the polynomial that we want to commit to, at point $\tau$, and multiply that by $g$, which will produce a single element from $\mathbb{G}$, and that is our commitment $c$. An amazing property to note here is that the polynomial can be of an arbitrarily large degree, but the commitment produced is still just a single element, i.e., a very short-sized piece of data, usually ~64 bytes.

## Open

This is where we prove an evaluation. To open the commitment at input $a \in \{0,\ldots,p-1\}$ to some value $b$, i.e.:

- Given an evaluation $\phi(a) = b$.
- the committer computes and outputs the proof $\pi = g^{q(\tau)}$. This is basically a witness polynomial calculation.
- $q(x) = \frac{(\phi(x)-b)}{x-a}$ is called the quotient polynomial. Such a $q(x)$ can exist only if $\phi(a) = b$. If this quotient polynomial exists, then it is a proof of evaluation. Mathematically it checks if the polynomial remainder theorem holds.

## Verify

This is where the evaluation proof is verified. A key property here is that verification can be done in constant time:

- Given a commitment $c = g^{\phi(\tau)}$, an evaluation $\phi(a) = b$ and the proof $\pi = g^{q(\tau)}$.
- The verifier checks that $e(c.g^{-b}, g) = e(\pi, g^{\tau}.g^{-a})$ where $e$ is a non-trivial bilinear mapping.

Note that the verifier knows the commitment $c$, evaluation $b$ , proof $\pi$, input $a$ and $g^{\tau}$. Values $c$, $b$, and $\pi$ are provided by the prover. Input $a$ is the opening query, which is provided by the verifier. $g^{\tau}$ is an entry in the SRS. For verification, only $g^{\tau}$ and $g$ are needed. This is why SRS is sometimes called the proving key and $(g, g^{\tau})$ are called the verification key. The verifier only needs the verification key to verify the proofs, and the prover uses the entire SRS as a proving key to generate the proofs.

We can visualize the Kate PCS in *Figure 18.9*:

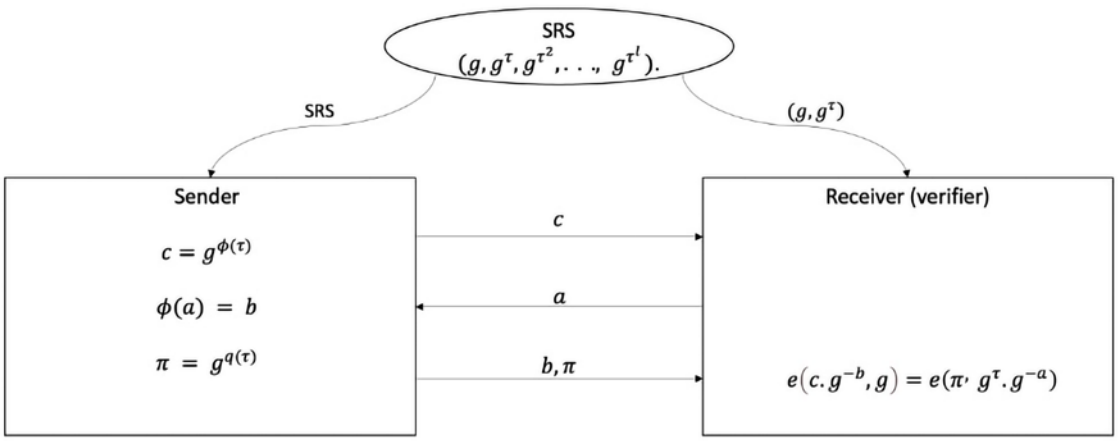


*Figure 18.9: A high level view of KGZ PCS*

If we have large amounts of data, how can we commit to it in such a way that a short message represents that entire dataset? In other words, we should be able to commit succinctly to that data. Secondly, we then should be able to prove values at specific locations.

This is where KZG is useful; we can represent large data as a polynomial and then evaluate it at secret points. Now you can see how PCS can be useful in blockchain. As the commitment of an arbitrarily large degree polynomial can be represented by just one group element, we can post just that commitment on the chain without needing to post everything to the blockchain, thus saving storage, bandwidth, and gas costs. In a rollup system a computation (e.g., state transition) can be represented as polynomials which the prover commits to and post on chain for verification. The verifier can ask the committer (e.g., the rollup provider) for evaluation on some random points; if the evaluations turn out to be correct, then it means that the entire computation is correct.

Ok, so now, after applying the commitment scheme we have achieved a protocol that allows us to commit to a polynomial with a short string. This means that it is a succinct protocol, but it is still interactive.

In order to make it non-interactive, we can use the **Fiat-Shamir** (**FS**) heuristic to transform this interactive protocol into a non-interactive one. FS transformation is a technique that is commonly used to transform an interactive protocol into a non-interactive one. The idea behind FS transformation is that the prover generates random bits on its own. Interactive protocols have a verifier that generates random bits and sends them to the prover, but non-interactive proof schemes transformed via FS enable the prover to generate random bits on their own. The requirement here is to provide enough uniform randomness to satisfy the soundness property of ZKPs. The interaction is replaced with access to a non-interactive random oracle. In practice, this is instantiated using a cryptographic hash function with carefully chosen inputs. After applying FS transformation we finally get a succinct non-interactive protocol, i.e., a SNARK.

There are different techniques to build the backend of a SNARK, with different pros and cons. For example, linear PCP-based SNARKs have the benefits of shortest proof and fastest verifier times. However, they need a circuit-specific setup, have slow prover times, and are not post-quantum resistant. A key example of this type is Groth16. Polynomial IOPs combined with KZG PCS produce SNARKs, which have the advantage of a universally trusted setup; however, the proofs are larger, the prover time is also slower, and they are also not post-quantum resistant. Some examples in this category are PLONK and Marlin. Linear PCP and constant-round PIOPs produce SNARKs that require a trusted setup. Combining a PIOP with FRI PCS produces proofs that are the shortest and post-quantum resistant; however, the prover time is slow and the proof size is still quite large.

This is how a SNARK is created at a high level. ZKPs is a huge subject and not everything can be covered in a single chapter. For further exploration, an excellent online resource is `https://zkp.science`, which can provide more insight into this fascinating subject.

ZK-SNARKs have many applications in the blockchain world. They can be used to improve the scalability of blockchain by creating a SNARK-based rollup. They can also be used to provide transaction privacy by obfuscating the transaction amounts and sender/receiver information.

Now let's see an example where we see these concepts in practice.

## Example

For this example, we'll use Zokrates (`https://zokrates.github.io/introduction.html`), which is an open source toolkit that allows users to build and deploy ZKP systems. It includes a high-level programming language called Zokrates DSL, which is designed to write programs that can be verified with zero-knowledge. To use Zokrates, programmers first write the ZKP program using the Zokrates DSL. It is then compiled into a low-level arithmetic circuit using the Zokrates compiler.

In this example the prover will prove to a verifier that they know a value `x`, which, when used to solve a simple equation, results in an answer that the prover already knows. Let's assume the equation is:

$$y = x^2 + 2$$

The condition here is that if $y$ is greater than or equal to 11, then it's a proof that the prover knows the value of $x$, without revealing the value of $x$.

So, let's write a simple program in Zokrates to do that:

```
def main(private field x) -> bool {
    bool y = if x*x + 2 >= 11 { true } else { false };
    return y;
}
```

Here we are simply defining a main function with a private field $x$. We declare it as private because we do not want to reveal this input but still prove that we know this value $x$.

Next we have the check that if the square of value of $x$ and adding 2 to it, is greater than or equal to 11 then variable $y$ being a bool, will be assigned with true; otherwise, it's false. In the next statement, we return $y$, which will be either true or false depending on the value of $x$. If the prover knows a legitimate value of $x$, that results in a value that is greater than or equal to 11 then they can prove to the verifier that they indeed know such a value of $x$ that satisfies the equation; otherwise, the verifier will be convinced that the prover doesn't know the value of $x$, which if plugged into the equation results in a value greater than or equal to 11.

We will use Remix IDE and Zokrates plugin to demonstrate this. Let's start:

1. First open Remix IDE.
2. On the home page, there is a feature plugin view; in that, find Zokrates and install it in Remix. Once it's installed you will notice that it's available in the left-hand side column on Remix IDE, as shown below in *Figure 18.10*:

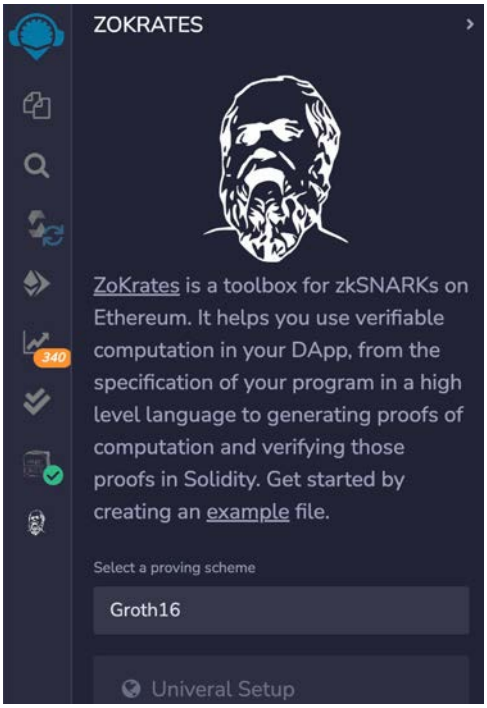


*Figure 18.10: The Zokrates toolkit in Remix IDE*

3. Once Zokrates is available in the Remix IDE, create a new file in it, and name it `main.zok`, as shown in *Figure 18.11*:


*Figure 18.11: The main.zok file*

4. Once the file is there, click on the Zokrates icon on the left-hand side, and click on **Compile** as shown below. If all is correct, it will compile the file. This compilation is equivalent to the creation of the circuit as discussed earlier.

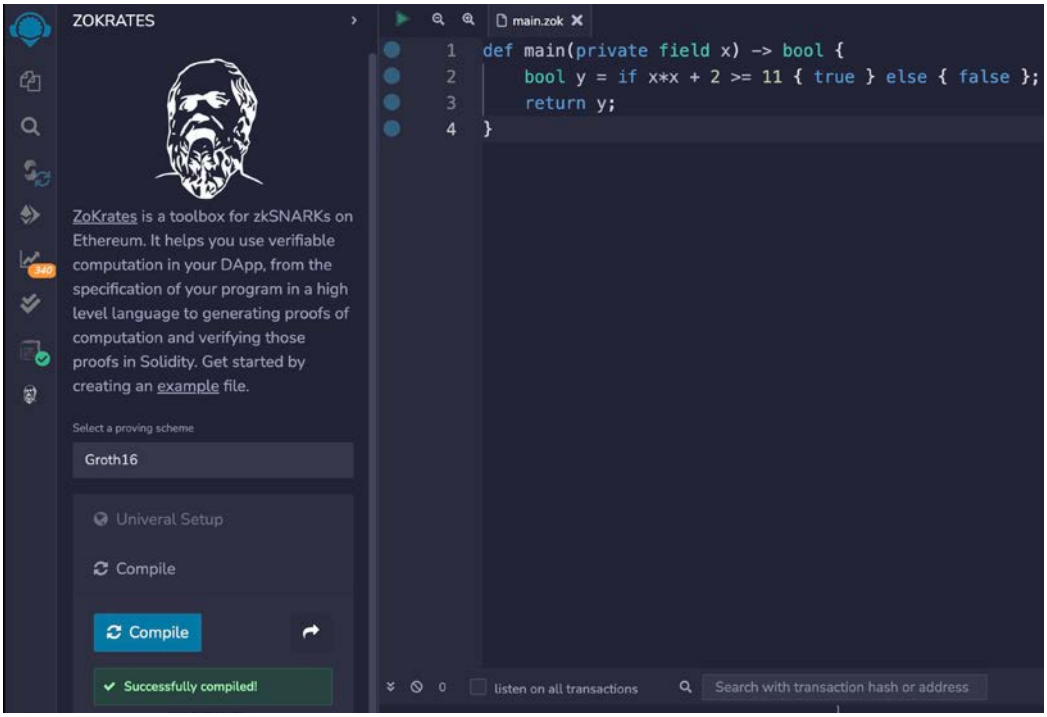


*Figure 18.12: Zokrates compilation*

5. The next step is to compute the witness, which can be achieved by inputting the value of $x$ and clicking on **Compute**. All these options are available under the Zokrates plugin, as shown below.

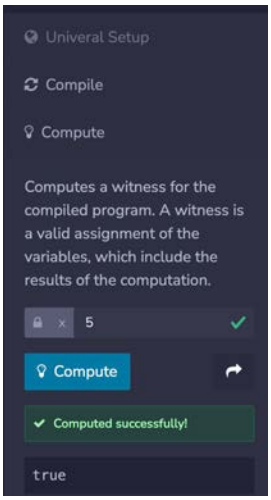


*Figure 18.13: Compute witness*

6. Now run the setup, which creates a proving key and a verification key as shown in *Figure 18.14*:

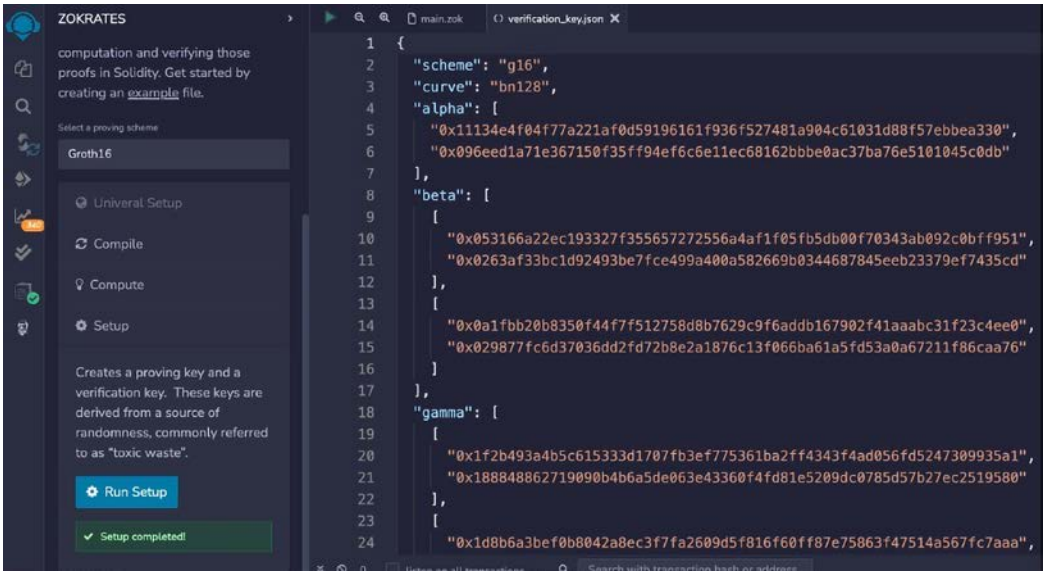


*Figure 18.14: Run Setup*

7. The next step is to generate the proof by using the roving key and witness generated in the previous steps. For this, click on **Generate** under the **Generate Proof** option in the Zokrates plugin, as shown below in *Figure 18.15*:

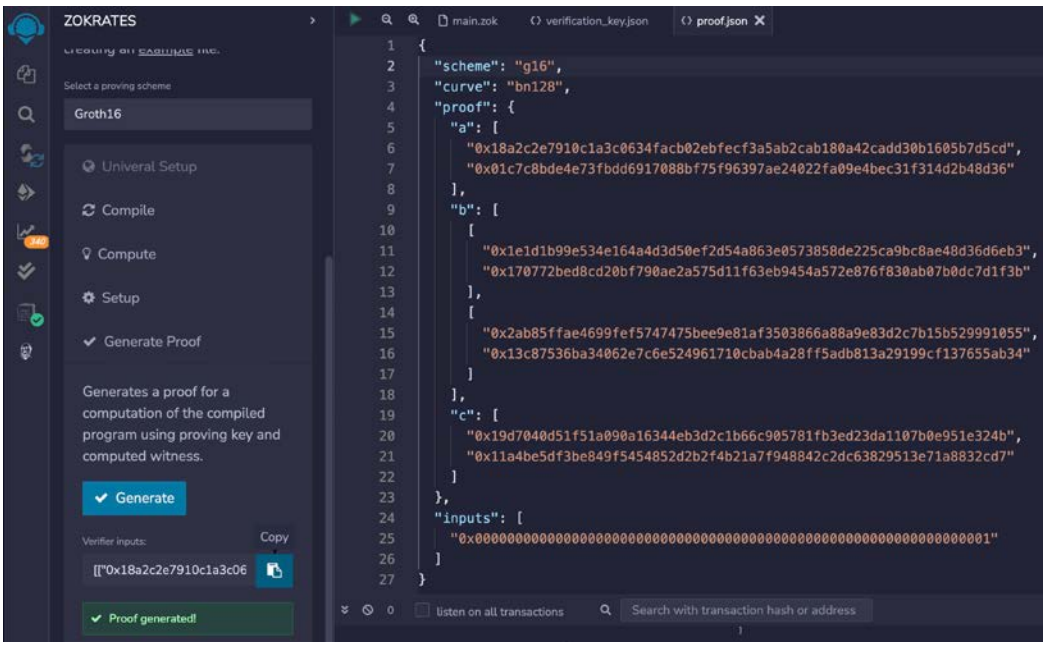


*Figure 18.15: Generate proof*

8. Once the proof is generated, it will generate a `proof.json` file containing the proof. Notice that on the left-hand column there is a **Verifier inputs** option; click on the **Copy** button to keep this proof in the clipboard or save it somewhere, because this is the input to the verifier smart contract that will be generated in the next step. We'll use this proof later on to pass it to the verifier contract for verification.
9. The next step is to generate the verifier; click on the **Export** button under the **Export Verifier** option to generate a solidity smart contract that contains a public function to verify the proof, i.e., the solution to the compiled Zokrates program. Note that this smart contract also contains the verification key. This is shown in *Figure 18.16* below:

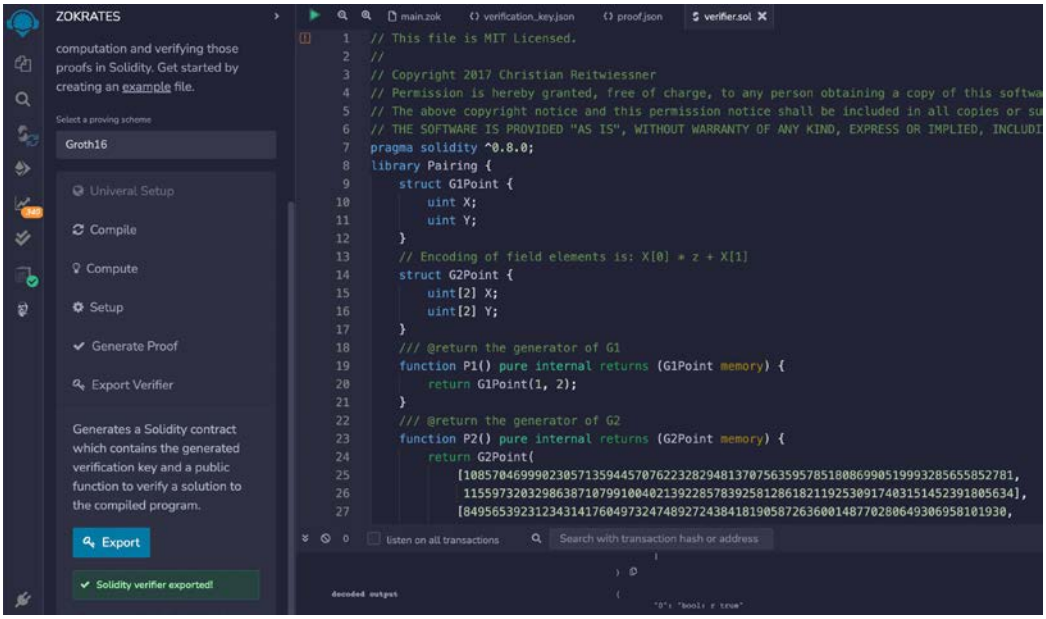


*Figure 18.16: The verification smart contract in Solidity*

This contract file name is `verifier.sol`. Now we can compile it and deploy it on the blockchain, either a *testnet* or *mainnet* Ethereum, as we learned in the previous chapters. If you want to deploy on the *mainchain* you can use MetaMask and select the **Injected web3** option in Remix IDE, as you learned before in the previous chapter.

Here we'll just use the Remix VM and run this example:

1. Now go to the **Solidity Compiler** option, and compile `verifier.sol`.
2. Under **DEPLOY & RUN TRANSACTIONS**, select `verifier.sol` under **CONTRACT** and click on **Deploy**.
3. Once deployed, simply invoke the `verifyTx` public function, by passing on the two parameters and clicking on the **call** button, as shown in the following screenshot in *Figure 18.17*:

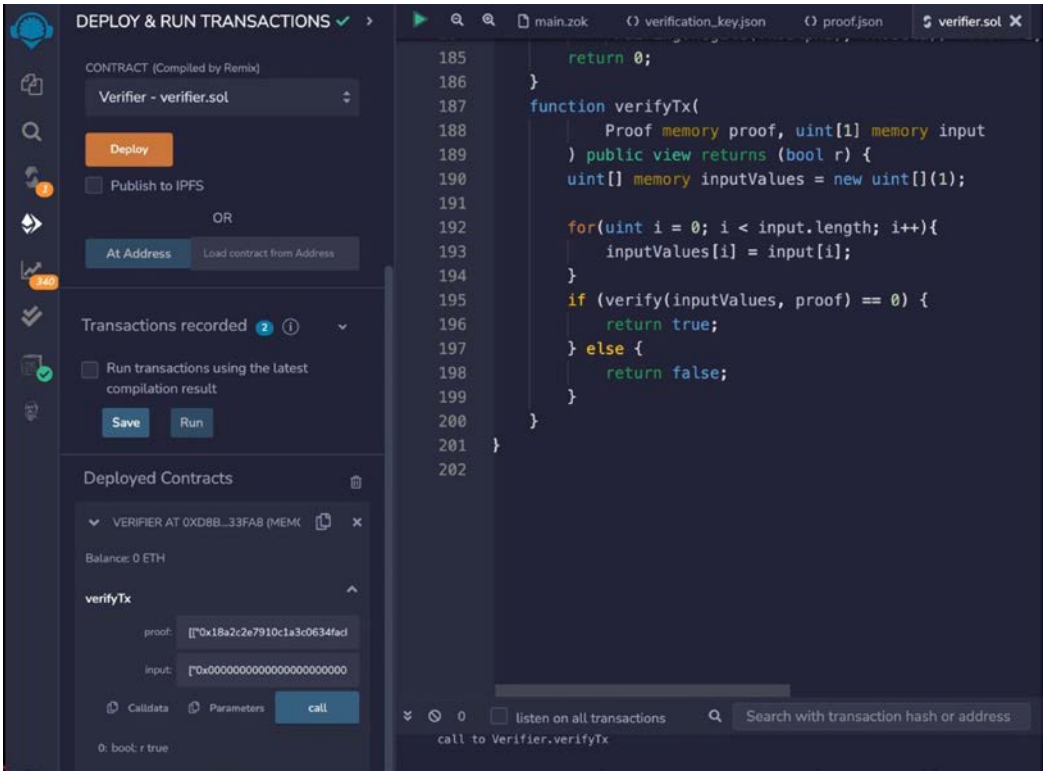


*Figure 18.17: Invoking the verifier smart contract's verityTx public function*

Notice that in the screenshot above, the `bool: true` value is returned, indicating that the verifier has been able to verify the proof. Remember that we did not reveal the value of $x$ at all to the verifier contract but simply passed two parameters, the proof tuple and the input value, as shown below:

```
[["0x18a2c2e7910c1a3c0634facb02ebfecf3a5ab2cab180a42cadd30b1605b7d5cd",
"0x01c7c8bde4e73fbdd6917088bf75f96397ae24022fa09e4bec31f314d2b48d36"],
[["0x1e1d1b99e534e164a4d3d50ef2d54a863e0573858de225ca9bc8ae48d36d6eb3",
"0x170772bed8cd20bf790ae2a575d11f63eb9454a572e876f830ab07b0dc7d1f3b"],
["0x2ab85ffae4699fef5747475bee9e81af3503866a88a9e83d2c7b15b529991055",
"0x13c87536ba34062e7c6e524961710cbab4a28ff5adb813a29199cf137655ab34"]],
["0x19d7040d51f51a090a16344eb3d2c1b66c905781fb3ed23da1107b0e951e324b",
"0x11a4be5df3be849f5454852d2b2f4b21a7f948842c2dc63829513e71a8832cd7"]],
["0x0000000000000000000000000000000000000000000000000000000000000001"]
```

With this we complete our introduction to Zokrates, and how programs can be built and proven with zero knowledge. Note that we can write the same program in another way, as shown below, where we use the `assert` statement instead of the `if else` structure:

```
def main(private field x) -> bool {
    assert (x*x + 2 >= 11)
    return;
}
```

I'll leave this as an exercise for the reader, where you can compile and run this program using the Zokrates plugin and Remix IDE.

Now that we know how Zokrates works and how we can build zero-knowledge solutions, can you build a program that proves to a verifier with zero knowledge that you are over 18 years of age without revealing exactly how old you are? It's a simple example but can be used in various scenarios, for example, proving that someone is eligible to drive a car, without revealing their exact age or date of birth.

Overall great progress has been made in the research and development of zero-knowledge proofs, but there are still some challenges. For example, the key theme that's emerging is that while succinctness and universality are increasing, the prover speed is not improving much. High-end hardware is required to generate proofs in a reasonable time; this is the reason why rollup providers tend to be high-end machines on the cloud. Some have even suggested ASICs and FPGAs for this purpose.

If in the future somehow, prover times can be decreased while keeping or making the verifier times and proof sizes smaller, it would further improve user adoption and enable use cases where comparatively low-end consumer hardware, like mobile phones, would be able to generate proofs without affecting user experience. Quantum resistance is another challenge; while STARKs being quantum-resistant, this is another interesting area of research to develop more techniques. Some research can be conducted into building new polynomial commitment schemes that are even more efficient. Formal verification and ensuring that the security properties of these protocols are correctly designed and implemented is another subject of keen interest. Imagine a DeFi protocol handling billions of dollars' worth of digital assets; if I have formal proof that the zero-knowledge system implemented in DeFi will work correctly in all cases, it will increase consumer confidence. This is also applicable generally to not only any ZK protocols but also protocols in blockchain and software, which we'll discuss in more detail in the next chapter, *Chapter 19*, *Blockchain Security*.

## Summary

This chapter covered blockchain privacy – a subject of profound importance. We covered what privacy is, its different facets, and how can we solve the privacy problem on blockchain. We also saw major technologies, such as ZKPs and particularly SNARKs, which are the most advanced and used form of ZKPs to solve privacy problems on the blockchain.

Moreover, we explored how Zokrates can be used to create zero-knowledge programs to complement the theoretical concepts, covered earlier in this chapter. Regarding privacy, we have many solutions, including but not limited to Zcash, Manta, Aleo, ZKOPRU, Aztec, and Espresso. To solve scalability, some key solutions are zkSync, Scroll, and Polygon Hermez.

In the next chapter we will introduce security, which is another challenge that blockchain technology faces.

## Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/ips2H`